A Reinforcement Learning Approach to the Andrews-Curtis Conjecture

Nikhil Pandit [1]

Abstract

We implement a reinforcement learning agent to simplify presentations of the trivial group into the trivial presentation. The methods closely mirror DeepCubeA (Agostinelli et al., 2019) and consist of Deep Approximate Value Iteration (DAVI) to train a cost-to-go function, together with A^*-search. Given presentations that have been modified from the trivial one by a sequence of 20-25 moves, our agent is able to find an unscrambling path in about 50% of cases, using only very limited compute resources.

1. Introduction

This project investigates the *Andrews-Curtis conjecture*, a conjecture in group theory that says that any presentation of the trivial group of the form

$$G = \langle g_1, \dots, g_n \mid r_1, \dots, r_n \rangle$$

which is "balanced" (i.e. has the same number of generators and relations) can be turned into the trivial presentation through a finite sequence of "moves":

(1) (Inversion) $r_i \mapsto r_i^{-1}$

(2) (Multiplication) $r_i \mapsto r_i r_j$ $(j \neq i)$

(3) (Conjugation) $r_i \mapsto g_j^{\pm 1} r_i g_j^{\mp 1}$ $(1 \leq j \leq n)$.

This problem is of interest in topology and group theory because it is a simple case of the (NP-hard) word problem in group theory, and has links to four-dimensional topology.

The Andrews-Curtis conjecture is generally believed to be false. There are many proposed counterexamples to the conjecture, namely presentations which are known to be equivalent to the trivial group, but for which it is not known whether a sequence of Andrews-Curtis moves exists. Here, we aim to train an agent to eliminate proposed counterexamples by showing that they can actually be trivialized by a sequence of Andrews-Curtis moves.

[1]Department of Mathematics, Stanford University. Correspondence to: Nikhil Pandit <npandit0@stanford.edu>.

1.1. Previous work

Previous work on counterexamples to the Andrews-Curtis conjecture included the use of genetic algorithms (Miasnikov, 1999) to rule out counterexamples to the conjecture with small relator lengths. Later work also included breadth-first search on the presentation tree (Havas & Ramsay, 2003), which suggested that an approach involving tree search is viable compared to the genetic algorithms approach. Our strategy lies more in this vein, but instead of using breadth-first search, we use A^*-search with a trained heuristic function.

1.2. Problem statement

We restrict ourselves to the case $n = 2$ in the Andrews-Curtis conjecture, so our groups have the form $\langle a, b \mid v, w \rangle$ where v and w are words in a, b, a^{-1}, b^{-1}, namely strings of the four symbols a, b, a^{-1}, b^{-1} in which consecutive appearances of a, a^{-1} (or b, b^{-1}) cancel out.

Task:

Given a group presentation

$$G = \langle a, b \mid v, w \rangle,$$

where v, w are words in a, b, a^{-1}, b^{-1}, obtain the trivial presentation

$$G = \langle a, b \mid a, b \rangle$$

through a sequence of the following moves:

Moves:

(1) (Swap) $(v, w) \mapsto (w, v)$

(2) (Inversion) $v \mapsto v^{-1}$

(3) (Multiplication) $v \mapsto vw$

(4) (Conjugation) $v \mapsto a^{\pm} v a^{\mp}$ or $v \mapsto b^{\pm} v b^{\mp}$.

Our methods closely follow the methods of (Agostinelli et al., 2019) in solving the Rubik's cube, and consist of three main steps:

(1) Devise a suitable state representation for group presentations.

(2) Automatically generate many representations of the trivial group.

(3) Train a *cost-to-go* function, i.e. a neural network which inputs a state and outputs an estimate of the number of moves necessary to solve the state.

(4) Perform A^*-search on the input states using the trained cost-to-go function. These steps are described in more detail in the following section.

2. Methods

2.1. State representations

We choose a large number L and create a state representation of a presentation $G = \langle a, b \mid v, w\rangle$ in the following way:

(1) if v or w has length $> L$, return the zero tensor of shape $2 \times 4 \times L$;

(2) otherwise, pad v, w to length L and join the one-hot encodings of v, w into a $(2 \times 4 \times L)$-shaped tensor.

The main drawback of this state representation method is that it prevents us from finding solutions which pass through intermediate presentations with words of length $> L$. For comparison, state representations of the Rubik's cube do not have this problem, as they naturally lie in a finite-dimensional space, whereas here the state space is infinite-dimensional.

2.2. Data generation

To train our cost-to-go function, we need to automatically generate many different presentations of the trivial group. We do so by repeatedly applying the above moves to the trivial group. (In analogy to the Rubik's cube setting, we call these presentations *scrambles* of the trivial presentation.) However, notice that of the seven allowable moves (SWAP, INV, MULT, and four conjugation moves), all but the first modify only the first word. The result is that randomly applying moves tends to produce presentations $G = \langle a, b \mid v, w\rangle$ with large disparities in the lengths of v and w. Therefore, we generate a scramble with n_{steps} steps according to the following algorithm:

Algorithm 1 Scramble generation

$P \leftarrow$ trivial presentation
for $n = 1$ to n_{steps} **do**
  if len$(P.v) > 2*$len$(P.w)$ **then**
    action$\leftarrow$SWAP
  else
    action$\leftarrow$random_action
  end if
  $P \leftarrow P$.apply(action)
end for

Finally, we generate a batch of data by repeatedly sampling $n_{\text{steps}} \sim \text{Unif}(1, \text{max\_steps}))$ and generating a scramble of length n_{steps} as above. However, we do not want to include any scrambles with words of length $> L$:

Algorithm 2 Data generation

$B \leftarrow$empty list
while len(B) <max_steps **do**
  $n_{\text{steps}} \sim \text{Unif}(1, \text{max\_steps})$
  scramble $\leftarrow$ get_scramble(n_{steps})
  if length(scramble.v)$\leq L$ and length(scramble.w)$\leq L$ **then**
    B.insert(scramble)
  end if
end while

In order for this data generation algorithm to work, L needs to be sufficiently large compared to max_steps; otherwise, the algorithm will discard most of the scrambles generated with a large number of steps. We concretely chose $L = 100$, max_steps$= 25$: Figure 1 shows that this choice of L is sufficiently large, as there is no reduction in the frequency of long scrambles.

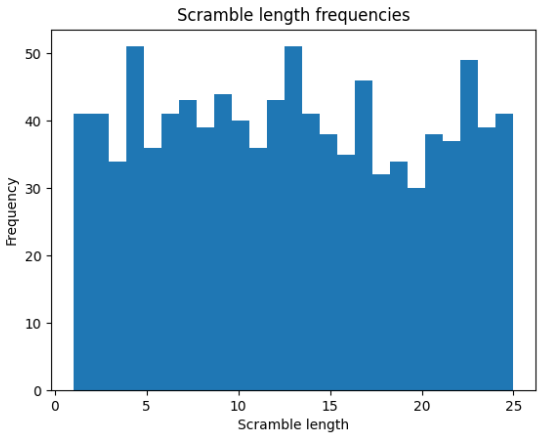

Figure 1. Frequencies of scramble lengths in a batch of generated presentations. Here the padding length is $L = 100$ and the maximum number of scramble moves is max_steps$= 25$. As seen in the figure, there is no drop-off in the frequency of long scramble lengths as compared to short ones.

2.3. Deep Approximate Value Iteration

The cost-to-go function J_θ, which is meant to estimate the number of steps from a given state to the goal state, is trained exactly as in (Agostinelli et al., 2019), by training on the L^2-loss between J_θ and bootstrapped predictions. We define a modified J_θ by

$$\tilde{J}_\theta(s) = \begin{cases} 0 & s \text{ trivial} \\ \infty & s \text{ out of bounds} \\ J_\theta(s) & \text{else.} \end{cases} \tag{1}$$

Then we train as follows:

Algorithm 3 Deep Approximate Value Iteration

$\theta \leftarrow$ initialize_parameters()
$\theta_e \leftarrow \theta$
for $m = 1$ to M **do**
  $X \leftarrow$scrambled_states(batch_size,max_scrambles)
  for $x_i \in X$ **do**
    $y_i \leftarrow 1 + \min_a \tilde{J}_\theta(a(s))$
  end for
  θ, loss $\leftarrow$ train(J_θ, X, y)
  if $M \equiv 0$ mod C and loss$< \epsilon$ **then**
    $\theta_e \leftarrow \theta$
  end if
end for

The bootstrapped predictions are made using a frozen copy θ_e of the current parameters θ, and these are reset to the true parameters θ every C training steps.

We defined J_θ as a deep neural network with two hidden layers of size 150. The training curve for the original naïve value iteration is shown in Figure 2.

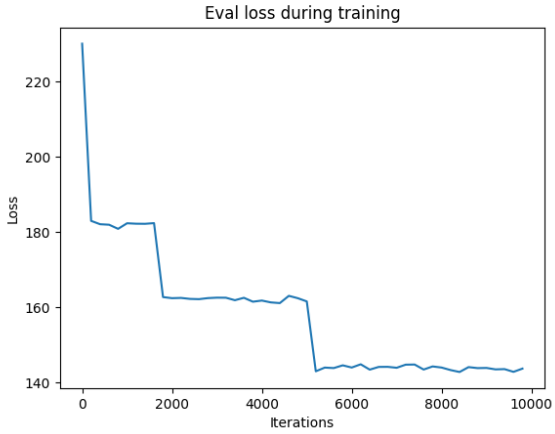

Figure 2. Mean squared error loss between the trained function J_θ and the number of steps used to generate each scramble.

Figure 2 should be interpreted with caution, as it plots the mean squared errors between two quantities that are not quite comparable. On the one hand, the function J_θ is trained to predict the length of the *shortest* solve of a scramble, whereas we are considering its MSE loss against the number of steps used to generate each scramble. The number of steps used to generate each scramble is always more than the length of the optimal solve of the scramble, but these may be different.

The sudden drops in loss in Figure 2 may seem somewhat odd. One explanation for the drops is that each drop occurs when the training encounters a new short presentation that it hasn't seen before. To test this hypothesis, we tried progressively increasing the scramble lengths seen by the model: first, it would see only scrambles of length ≤ 5, then scrambles of length ≤ 10, and so on until it reached scrambles of max length 25. The results are shown in Figures 3 through 7. Indeed, this technique produces a significant improvement: overall, the loss tends to go down during training (although it is more variable for the intermediate stages max_scrambles$= 10, 15$), and the final value of the loss is around 40, much better than the value of ~ 140 obtained from using long scrambles from the beginning.

2.4. A^*-search

The final piece of the approach is to use A^*-search to find a shortest path between a given presentation and the goal state (the trivial presentation). In A^*-search, we initialize a search tree rooted at our presentation. Each node x of

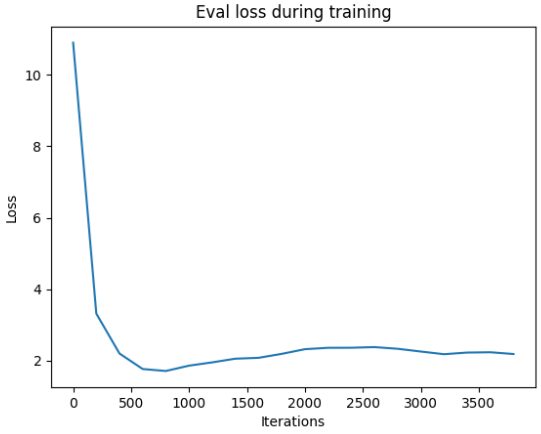

Figure 3. max_scrambles= 5

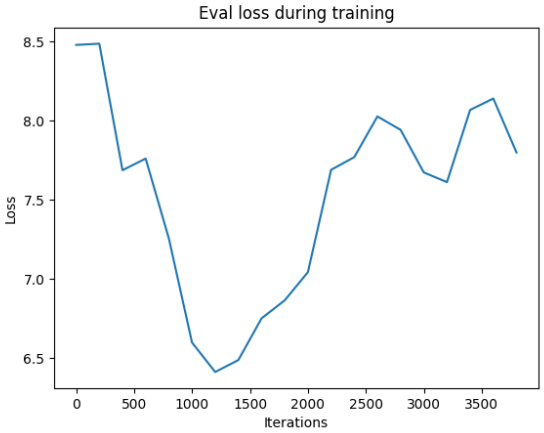

Figure 4. max_scrambles= 10

Figure 5. max_scrambles= 15

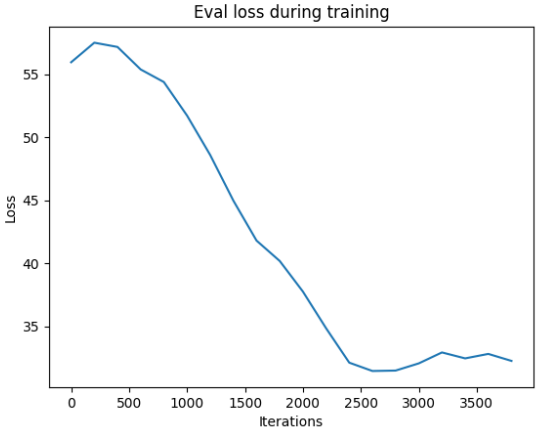

Figure 6. max_scrambles= 20

Figure 7. max_scrambles= 25

the tree has a value $g(x)$ (the distance from the root) and $h(x)$ (a heuristic for the approximate distance to the goal). At each stage of the search, we select the node minimizing $g(x) + h(x)$ and expand the tree by adding all of the node's descendants, computing g and h for each new node as we go.

In our application, the heuristic $h(x)$ is exactly the modified cost-to-go function $\hat{J}_\theta(x)$ from equation (1).

3. Results

We evaluated two kinds of search:

(1) Greedy search, in which we always take the action resulting in the smallest value of the heuristic $\hat{J}_\theta$;

(2) A^* search with $\hat{J}_\theta$.

For each type of search, we plot the proportion of scrambles that the agent was able to solve as a function of scramble length, and also show the average length of the solutions that the agent found.

The results of greedy search are shown in Figures 8 and 9. Figure 8 shows that greedy search performs surprisingly well, managing to solve a nontrivial number of scrambles even above 20 moves. (By comparison, every Rubik's cube scramble can be solved in 20 moves.) However, the success of greedy search seems to suffer a linear decay with scramble length.

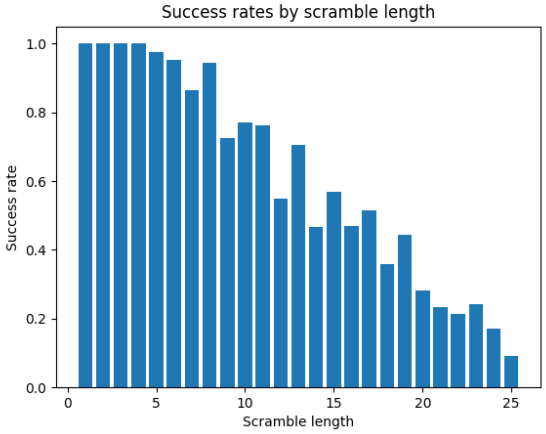

Figure 8. **Greedy search:** Success rates on various scramble lengths. The algorithm was successful if it could reach the goal state within 100 steps.

Figure 9 shows that the greedy search is also finding relatively efficient solutions: most of its solutions are below 16 moves, showing that the 100-move limit on the greedy search is not really significant.

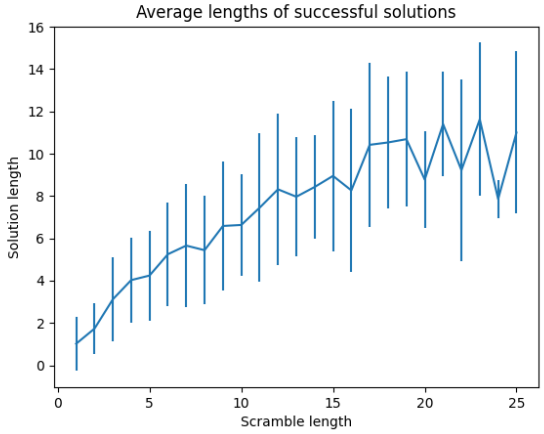

Figure 9. **Greedy search:** This plot shows the solution lengths found by the agent on scrambles that it was able to solve.

Figures 10 and 11 show the performance of A^*-search.

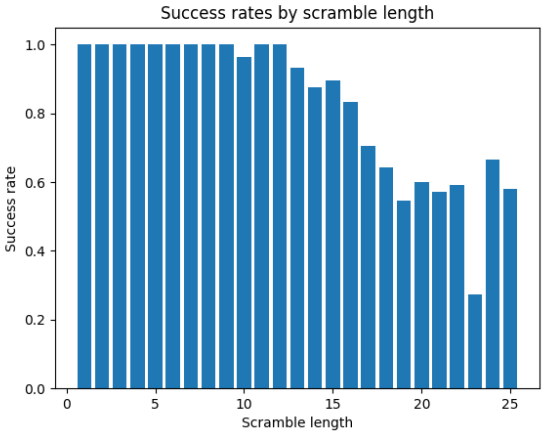

Figure 10. ***A**-search:** Success rates on various scramble lengths. The algorithm was successful if it could reach the goal state within 1000 tree expansions.

Figure 10 demonstrates near-perfect success of A^* search at resolving scrambles under 12 moves, and moderate success at resolving even long scrambles, solving 67% of 24-move scrambles and 58% of 25-move scrambles.

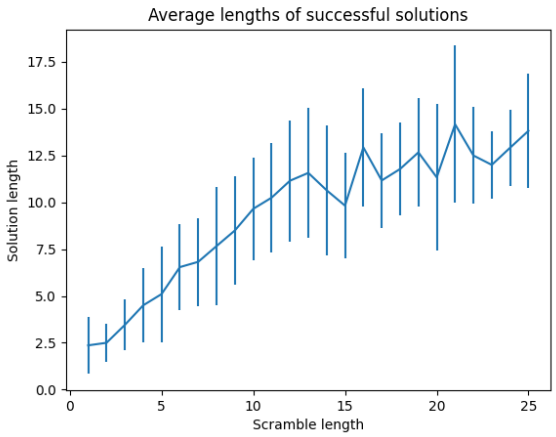

Figure 11. ***A**-search:** This plot shows the solution lengths found by the agent on scrambles that it was able to solve.

Finally, we examined the performance of our agent on the special group presentation

$$G = \langle a, b \mid abab^{-1}a^{-1}b^{-1}, b^3a^{-2} \rangle,$$

which was a long-standing potential counterexample to the Andrews-Curtis conjecture until it was resolved by genetic algorithm methods in (Miasnikov, 1999). Unfortunately, despite allowing the A^*-search to examine up to 100,000 nodes, the agent was not able to find a solution for this special group presentation.

4. Discussion

The three most salient results of the experiments above are that

(1) when training with bootstrapping as in DAVI, arranging training samples in increasing order of complexity drastically improves training;

(2) greedy search performs near-perfectly for scrambles of length ≤ 4, then decays roughly linearly to a success rate of about $10 - 20\%$ for scrambles of length $20 - 25$;

(3) A^*-search performs near-perfectly for scrambles of length ≤ 12, then decays to a success rate of about 50% for scrambles of length 20-25.

It is important to emphasize that A^*-search is not performing well simply by exhaustively searching a tree: there are $7^{12} = 1.4 \times 10^{10}$ possible sequences of 12 moves from the root node, but the search is limited to investigating just 1000 nodes. Hence, the cost-to-go function J_θ is crucial to making the search efficient.

In a similar vein, it is important to realize that with the strategy of training the cost-to-go function J_θ on progressively more complicated scrambles, there is overfitting/sample memorization on the first set of small scrambles (Figure 3), as there are only $7^5 = 16,807$ ways to apply 5 moves to the trivial group. However, there is no overfitting or sample memorization when we train on the more complicated scrambles; by the end, there are $7^{25} = 1.3 \times 10^{21}$ 25-move scrambles, way more than the 120,000 presentations that the agent sees in training.

5. Conclusion

The agent trained in this project is able to solve complicated scrambles with extremely limited compute resources (training and search were done on a CPU). However, with this limited compute, it is not yet able to eliminate proposed counterexamples to the Andrews-Curtis conjecture. One potential reason for this failure is that the types of presentations the agent sees in training involve long relations that can be systematically undone, while proposed counterexamples tend to be compact to express but difficult to simplify: any simplification requires the agent to pass through much more complicated intermediate presentations.

Further research in this direction should:

(1) implement the methods of this project with larger compute resources;

(2) modify the A^*-search used here to a batched weighted A^*-search, as described in (Agostinelli et al., 2019)

(3) curate the data generation process to put higher weight on short presentations which nevertheless cannot be solved in few moves.

6. Code

Code for this project can be accessed from the Github repository npandit0/Andrews-Curtis.

References

Agostinelli, F., McAleer, S., Shmakov, A., and Baldi, P. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1 (8):356–363, August 2019. ISSN 2522-5839. doi: 10.1038/s42256-019-0070-z.

Havas, G. and Ramsay, C. Breadth-first search and the andrews–curtis conjecture. *International Journal of Algebra and Computation*, 13(01):61–68, February 2003. ISSN 0218-1967. doi: 10.1142/S0218196703001365.

Miasnikov, A. D. GENETIC ALGORITHMS AND THE ANDREWS–CURTIS CONJECTURE. *International*

Journal of Algebra and Computation, 09(06):671–686, December 1999. ISSN 0218-1967, 1793-6500. doi: 10.1142/S0218196799000370.